Script **generated by TTT**

Title: Simon: Programmiersprachen (11.01.2013)

Date: Fri Jan 11 10:05:08 CET 2013

Duration: 87:15 min

Pages: 34

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

Mixins

Dr. Axel Simon and Dr. Michael Petter
Winter term 2012

“What advanced techiques are there besides multiple implementation inheritance?”

Outline

Weak implementation inheritance

1. Decorator Problem
2. Wrapper Problem

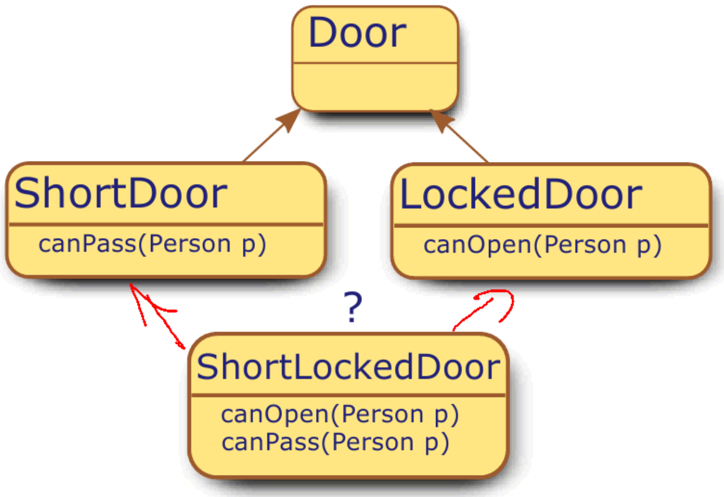
Inheritance in Detail

1. Models for single inheritance
2. Introducing Mixins
3. Modelling Mixins

Mixins in the wild

1. Mixins as C++-Pattern
2. Extension methods
3. Native Mixins

The Adventure Game



Door

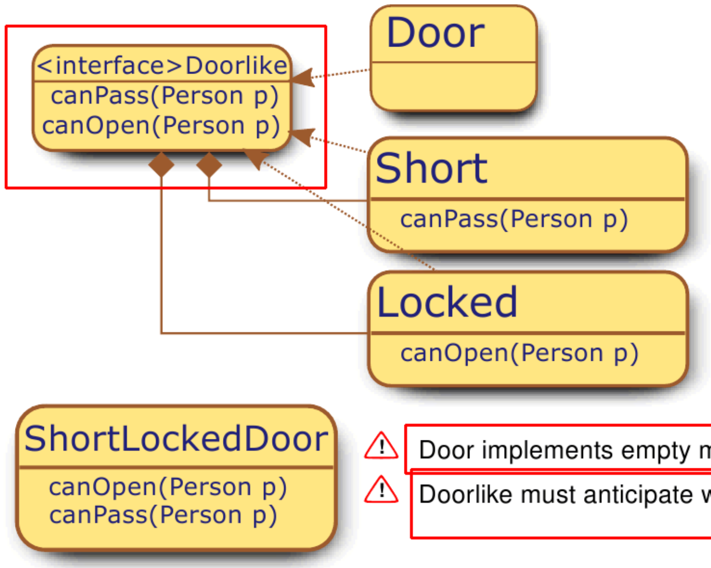
ShortDoor
canPass(Person p)

LockedDoor
canOpen(Person p)

?

ShortLockedDoor
canOpen(Person p)
canPass(Person p)

The Adventure Game



<interface>Doorlike
canPass(Person p)
canOpen(Person p)

Door

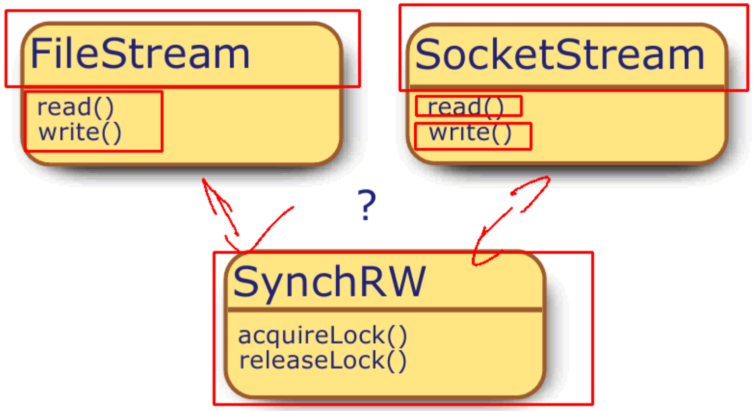
Short
canPass(Person p)

Locked
canOpen(Person p)

ShortLockedDoor
canOpen(Person p)
canPass(Person p)

- ⚠ Door implements empty methods
- ⚠ Doorlike must anticipate wrappers

The Wrapper



FileStream
read()
write()

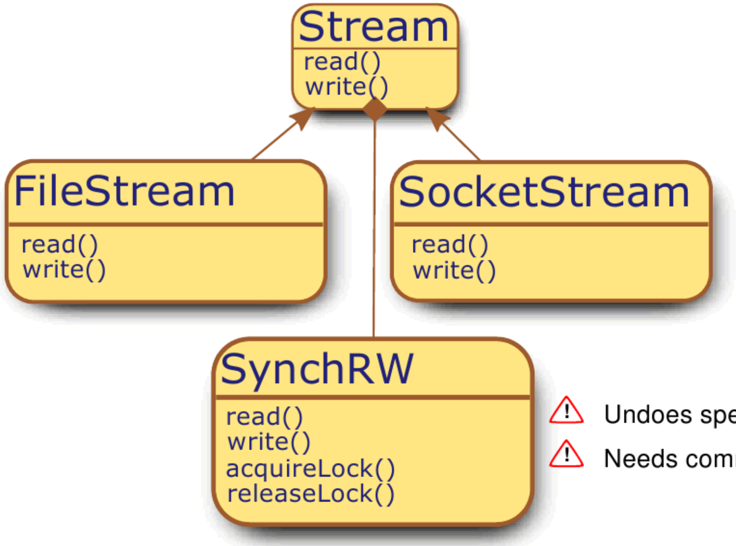
SocketStream
read()
write()

?

SynchRW
acquireLock()
releaseLock()

- ⚠ Cannot inherit from both seperately
- ⚠ Creating new wrapping Classes duplicates code

The Wrapper



Stream
read()
write()

FileStream
read()
write()

SocketStream
read()
write()

SynchRW
read()
write()
acquireLock()
releaseLock()

- ⚠ Undoes specialization
- ⚠ Needs common ancestor

“Let’s go back to the basics of inheritance”

Abstract model for Smalltalk-Inheritance

Smalltalk inheritance is the archetype for inheritance in mainstream languages like Java or C#.

- Types of Classes abstracted to maps from Identifiers to qualified methods
- Subtypes are specified as increments Δ to their parents
- `super` calls are delegated to the parent
- ⇝ Parent is connected to the increment as a parameter $\Delta(Parent)$
- Combination operator $\oplus$ merges operands, prefering the left argument

Smalltalk-like Inheritance is defined as $C = \Delta(P) \oplus P$

Example: `Doors`

$$Door = \{canPass \mapsto \bot, canOpen \mapsto \bot\}$$

$$\boxed{LockedDoor} = \boxed{(\{canOpen \mapsto \boxed{LockedDoor.canOpen}\}}\boxed{(Door})) \oplus \boxed{Door}$$

Excursion: Beta-Inheritance

Beta-style inheritance is designed to provide security from replacement of a method by a different method.

- methods in parent overwrite methods in subclass
- `inner` as keyword to delegate control to subclass (⇝ `super`)
- ⇝ parent arranges the exact spot, where the subclass can take over

Example (equivalent syntax):

```
class Person {
  String name ="Axel Simon";
  public virtual String toString(){ return name+inner(); };
};
class Graduate extends Person {
  public extended String toString(){ return ", Ph.D."; };
};
```

Beta-like Inheritance is defined as $C(\underline{inner}) = P(\Delta(\underline{inner})) \oplus \Delta(\underline{inner})$

⇝ Types in Beta are ⇝ *Lambda-Expressions*

Generalizing Beta- and Smalltalk-Inheritance

We introduce the combination operator, which joins attributes and performs `super`/`inner` bindings:

$$\boxed{A \triangleright B} = \boxed{A(B) \oplus B}$$

Smalltalk	$\boxed{C = \Delta \triangleright P}$
Beta	$\backslash\lambda.C(\lambda) = \backslash\lambda.\boxed{(P \triangleright \Delta(\lambda)})$

⇝ Both Systems differ only in the direction of growth (and the lambda-expression)

Excursion: CLOS-Inheritance

CLOS(Common Lisp Object System)-style inheritance offers multiple implementation inheritance featuring linearization.

- methods in childs overwrite methods in parents
- `super` as keyword to delegate control to direct parent (⇝ linearization)

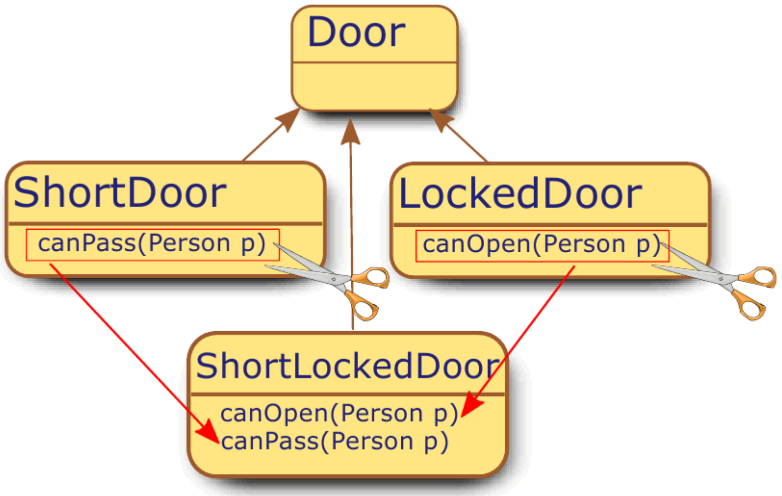
Example (equivalent syntax):

```
class Person {
  String name ="Axel Simon";
  public String toString(){ return name; }
}
class Graduate extends Person {
  public String toString(){ return super.toString()+", Ph.D."; }
}
class Doctor extends Person {
  public String toString(){ return "Dr. "+super.toString(); }
}
class ResearchingDoctor extends Doctor,Graduate {}
```

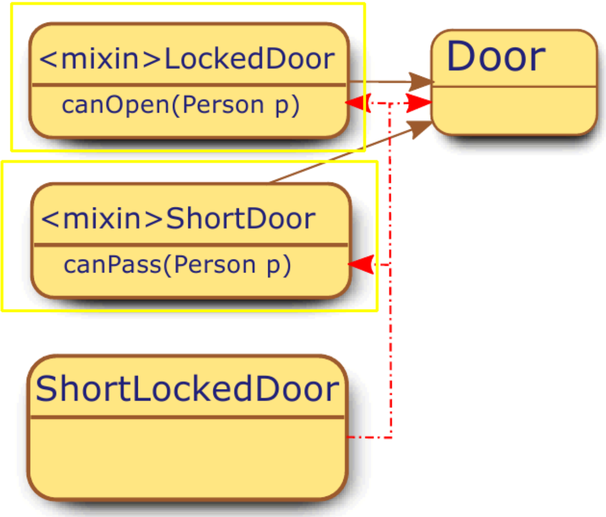
CLOS-like Multiple-Inheritance: $C = \Delta_1 \triangleright (\Delta_2 \triangleright (\ldots \triangleright P)\ldots)$

“So what do we really want?”

Adventure Game with Code Duplication



Adventure Game with Mixins



Adventure Game with Mixins

```
class Door {
 boolean canOpen(Person p) { return true; };
 boolean canPass(Person p) { return true; };
}
mixin Locked extends Door {
 boolean canOpen(Person p){
  if (!p.hasItem(key)) return false; else return super.canOpen(p);
 }
}
mixin Short extends Door {
 boolean canPass(Person p){
  if (p.height()>1) return false; else return super.canPass(p);
 }
}
class ShortDoor = Short(Door);
class LockedDoor = Locked(Door);
mixin ShortLocked = Short compose Locked;
class ShortLockedDoor  = Short(Locked(Door));
class ShortLockedDoor2 = ShortLocked(Door);
```

Abstract model for Mixins

Mixin Composition: $\backslash\lambda.(M_1 \star M_2)(\lambda) = \backslash\lambda.\ M_1(M_2(\lambda) \oplus \lambda) \oplus (M_2(\lambda) \oplus \lambda)$

Example: `Doors`

$$\begin{aligned}
Door &= \{canPass \mapsto Door.canPass, canOpen \mapsto Door.canOpen\}\\
Locked &= \{canOpen \mapsto Locked.canOpen\}\\
Short &= \{canPass \mapsto Locked.canPass\}\\
ShortLocked &= Short \star Locked = \backslash\lambda.\ Short(Locked(\lambda) \oplus \lambda) \oplus Locked(\lambda)
\end{aligned}$$

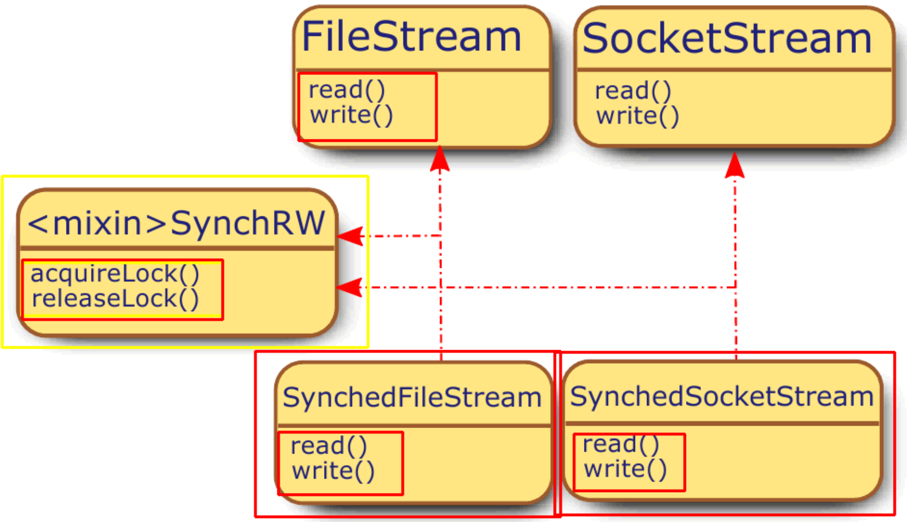
Standard classes are handled as *degenerated Mixins*, binding $\emptyset$ as composite reference; Mixins are connected to classes via inheritance $\triangleright$:

$$M(P) = M(P) \oplus P = M \triangleright P$$

Example: `Doors`

$$\begin{aligned}
ShortLockedDoor &= (Short \star Locked) \triangleright Door\\
&= (Short(Locked(Door) \oplus Door) \oplus Locked(Door)) \oplus Door
\end{aligned}$$

Wrapper with Mixins



Types of Mixins

Subtype Relation $\ll$:

- $$\frac{\bot}{T \ll \texttt{Object}}$$
- $$\frac{T_1 = T_2T_3}{T_1 \ll T_2 \wedge T_1 \ll T_3}$$
- $\ll$ Reflexively and transitively closed

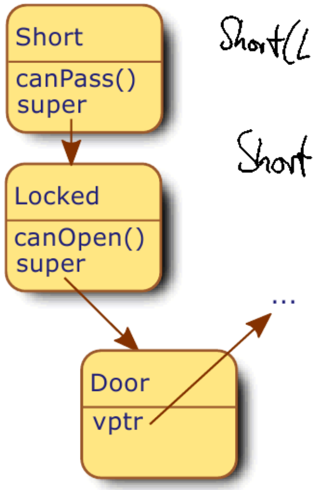
Example: `Doors`

$$\begin{aligned}
ShortLocked \ll Locked &\wedge ShortLocked \ll Short\\
ShortLockedDoor \ll Short &\wedge ShortLockedDoor \ll Locked\\
ShortLockedDoor \ll ShortLocked &\wedge ShortLockedDoor \ll ShortDoor
\end{aligned}$$

Implementing Mixins

```
class Door {
 boolean canOpen(Person p)...
 boolean canPass(Person p)...
}
mixin Locked extends Door {
 boolean canOpen(Person p)...
}
mixin Short extends Door {
 boolean canPass(Person p)...
}
class ShortDoor
   = Short(Door);
class ShortLockedDoor
   = Short(Locked(Door));
...

ShortDoor d
   = new ShortLockedDoor();
```



Short(Locked(Door))

ShortD

⚠ `super`-References not statically resolvable

Programming Mixins

There are different ideas to bring mixins into daily programming:

C++	Templates & Multiple Inheritance
C#	Extension Methods
Java	Aspect Orientation or Virtual Extension methods
Ruby/Python	Native mixins

"Surely multiple inheritance is powerful enogh to simulate mixins?"

Simulating Mixins in C++

```
template <class Super>
class SyncRW : public ~~Super~~{
  public: virtual int read(){
    acquireLock();
    int result = Super::read();
    releaseLock();
    return result;
  };
  virtual void write(int n){
    acquireLock();
    Super::write(n);
    relaseLock();
  };
  // ... acquireLock & releaseLock
};
```

Simulating Mixins in C++

```
template <class Super>
class LogOpenClose : public Super {
   public: virtual void open(){
    Super::open();
    log("opened");
   };
   virtual void close(){
    Super::close();
    log("closed");
   };
   protected: virtual void log(char*s) { ... };
};
class MyDocument : public SyncRW<LogOpenClose<Document>> {};
```

True Mixins vs. C++ Mixins

True Mixins

- `super` natively supported
- Mixins as Template do not offer composite mixins
- C++ Type system not modular
- ⇝ Mixins have to stay source code
- Hassle-free simplified version of multiple inheritance

C++ Mixins

- Mixins reduced to templated superclasses
- Can be seen as coding pattern

Common properties of Mixins

- Linearization is necessary
- ⇝ Exact sequence of Mixins is relevant

“So how about method extensions?”

Extension Methods (C#)

Central Idea:

Uncouple method definitions and implementations from class bodies.

Purpose:

- retrospectively add methods to complex types
- especially provide implementations for *interface methods*

Syntax:

1. Specify a static class with static methods
2. Explicitely specify receiver type as first first parameter with keyword `this`
3. Bring the carrier class into scope (if needed)
4. Call extension method in *infix form*

```
public class Person{
 public int size = 160;
 public bool hasKey() { return true;}
}
public interface Short {}
public interface Locked {}
public static class DoorExtensions {
 public static bool canOpen(this Locked leftHand, Person p){
  return p.hasKey();
 }
 public static bool canPass(this Short leftHand, Person p){
  return p.size<160;
 }
}
public class ShortLockedDoor : Locked,Short {
 public static void Main() {
  ShortLockedDoor d = new ShortLockedDoor();
  Console.WriteLine(d.canOpen(new Person()));
 }
}
```

Extension Methods as Mixins

Pro Extension Methods

- transparently extend arbitrary types
- for many cases offer enough flexibility

Contra Extension Methods

- Interface declarations empty, thus kind of purposeless
- Inherited properties always of higher priority then extensions
- Class-code is distributed over several class bodies
- Still no `super` reference

⚠ Limited scope of extension methods prohibits expected behaviour:

```
public interface Locked {
  public bool canOpen(Person p){
}
public static class DoorExtensions {
 public static bool canOpen(this Locked leftHand, Person p){
  return p.hasKey();
 }
}
```

Excursion: *Virtual* Extension Methods (Java 8)

Project *Lambda* from the upcoming Java version advances one pace further:

```
interface Door {
  boolean canOpen(Person p);
  boolean canPass(Person p);
}
interface Locked extends Door {
  boolean canOpen(Person p) default { return p.hasKey(); }
}
interface Short extends Door {
  boolean canPass(Person p) default { return p.size<160; }
}
public class ShortLockedDoor implements Short, Locked, Door {
}
```

Implementation

…consists in adding an interface phase to `invokevirtual`'s name resolution

⚠ Polymorphic Overwriting

Still, default methods can not overwrite abstract methods from abstract classes

---

"Ok, ok, show me a language with native mixins!"

Ruby

```
class Person
  attr_accessor :size
  def initialize
    @size = 160
  end
  def hasKey
    true
  end
end
class Door
  def canOpen
    true
  end
  def canPass(person)
    person.size < 210
  end
end
```

```
module Short
  def canPass(p)
    p.size < 160 and super(p)
  end
end
module Locked
  def canOpen(p)
    p.hasKey() and super
  end
end
```

```
class ShortLockedDoor < Door
  include Short
  include Locked
end
```

```
p = Person.new
d = ShortLockedDoor.new
puts d.canPass(p)
```

Lessons Learned

Lessons Learned

1. Formalisms to model inheritance
2. Mixins provide soft multiple inheritance
3. Multiple inheritance can not compensate `super` reference
4. (Virtual) extension methods migrate to major languages
5. Full extent of mixins only when mixins are 1st class language citizens

---

Further reading...

- Gilad Bracha and William Cook.
  Mixin-based inheritance.
  European conference on object-oriented programming on Object-oriented programming systems, languages, and applications (OOPSLA/ECOOP), 1990.
- Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black.
  Traits: A mechanism for fine-grained reuse.
  ACM Transactions on Programming Languages and Systems (TOPLAS), 2006.
- Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen.
  Classes and mixins.
  Principles of Programming Languages (POPL), 1998.
- Brian Goetz.
  Interface evolution via virtual extension methods.
  JSR 335: Lambda Expressions for the Java Programming Language, 2011.
- Anders Hejlsberg, Scott Wiltamuth, and Peter Golde.
  C# Language Specification.
  Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
  ISBN 0321154916.